Enabling Intermittent Computing on High-Performance Out-of-Order Processors

Sivert T. Sliper, Domenico Balsamo, Alex S. Weddell and Geoff V. Merrett
Centre for IoT and Pervasive Systems
School of Electronics and Computer Science
University of Southampton, UK
{sts1u16,db2a12,asw,gvm}@ecs.soton.ac.uk

ABSTRACT

Intermittent computing is a new paradigm enabling battery-less computing devices to be powered directly from energy harvesting, enabling IoT devices that are free from the cost, size and lifetime constraints of batteries. To cope with frequent power interruptions, intermittent computing systems save computational progress before power is lost, and restore it when power returns. Recent research in power-neutral operation of multiprocessor system-on-chips (MPSoCs), where performance scaling is used to instantaneously match power consumption with supply, motivates the need for intermittent computing on high-performance systems. Existing works provide solutions for microcontrollers, but with the increased complexity of high-performance SoCs, new challenges such as hierarchical memory and dependence on large existing libraries emerge. In this paper, we provide a taxonomy of published intermittent computing methods and identify the most suitable method for high-performance SoCs. The chosen method is then implemented and experimentally validated on an Arm A9 out-of-order application processor. Results show that state can be saved/restored correctly in 8.6 *ms* for a minimal bare-metal application, which is an order of magnitude faster than the platform's hardware boot time.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; *Embedded software*; *System on a chip*;

KEYWORDS

Internet of Things, Intermittent Computing, Battery-less Computing, Power-Neutral Computing

ACM Reference Format:
Sivert T. Sliper, Domenico Balsamo, Alex S. Weddell and Geoff V. Merrett. 2018. Enabling Intermittent Computing on High-Performance Out-of-Order Processors. In *ENSsys '18: International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems, November 4, 2018, Shenzhen, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3279755.3279759

1 INTRODUCTION

Powering IoT devices directly from energy harvesting (EH) sources removes the size, cost, maintenance and lifetime constraints imposed by energy storage devices such as rechargeable batteries or supercapacitors. Energy from ambient sources such as light, mechanical vibration, wind and temperature difference can be harvested indefinitely [5]. However, the power output from harvesters is generally unpredictable, uncontrollable and unstable [13]. To cope with the instability of EH, intermittent computing (IC) saves snapshots of system state to nonvolatile memory (NVM) [20]. When power returns after a power-failure, system state is restored from the snapshot and the computation continues. Thus, long-running computations can be sustained over several power cycles.

To extend the range of operation and better utilise harvested power, Balsamo et al. proposed the concept of power-neutral operation, whereby performance scaling is used to instantaneously match power consumption with harvested power [2]. Fletcher et al. extended this concept by implementing power-neutral operation on a heterogeneous multiprocessor system-on-chip (MPSoC) by using a combination of dynamic voltage and frequency scaling (DVFS) and dynamic power management (runtime enabling/disabling of CPU cores) [10]. Power-neutral systems have negligible energy storage, so must react quickly to variations in supply, and can only operate for a very short time if supply drops below minimum. Even for power-neutral systems, IC is needed to preserve progress through periods of insufficient power. IC was included in the microcontroller approach to power-neutral computing [2], but was not included in the method for MPSoCs [10] – likely because of the complexity and viability of implementing IC on the desktop Linux operating system used in their work and the lack of existing IC methods for high-performance systems.

Although IC is a relatively young field of research (the pioneering work, *Mementos* [20], was published in 2011), several fundamentally diverging approaches have been published, each with their own unique advantages and drawbacks. In this paper, we examine the current state of the art of IC and provide a taxonomy of existing methods to determine the most viable solution for complex high-performance computing systems, where dependence on large existing code bases is likely (Section 2). The selected IC method is implemented on an Arm A9 processing system [21] (Section 3). This is the first time software-based IC is implemented on a system more complex than a microcontroller, addressing new challenges such as handling several processor modes and control and acceleration co-processors. Techniques to tailor the system towards efficient IC are presented (Section 4). The system is experimentally validated and results analysed (Section 5). The contributions of this paper are:

- A taxonomy of existing works determining the optimal IC method for high-performance systems (Section 2);
- An evaluation on the viability of integrating IC into a power-neutral MPSoC approach (Section 2);

ENSsys '18, November 4, 2018, Shenzhen, China
2018. ACM ISBN 978-1-4503-6047-0/18/11…$15.00
https://doi.org/10.1145/3279755.3279759

- The first implementation of intermittent computing on a high-performance Out-of-Order processor (Section 3).

2 TAXONOMY OF EXISTING IC STRATEGIES

Reliable and efficient state retention through power cycles is the main priority in IC. Conflicting design goals such as having minimal hardware dependency [7, 16, 20], minimizing the number of snapshots written during a power cycle [3, 4, 7], reducing the size of snapshots [6, 7, 11, 14] and maintaining compatibility with existing codebases [3, 4] has driven research into fundamentally diverging approaches. In order to determine the most suitable approach for high-performance systems, we introduce a taxonomy based on the following three classes of IC strategies.

- **Static IC:** where snapshots are saved at predetermined (design-time or compile-time) checkpoints in the program.
- **Task-based IC:** where the application is divided into small tasks that are executed atomically by a runtime.
- **Reactive IC:** where snapshots are saved as a reaction to the environment.

The fundamental divergence between existing methods is between static and reactive IC. Task-based IC can be regarded as a recent development of static IC, where checkpoints are replaced by task boundaries. The classes are useful because each of the three classes present their own advantages and drawbacks. The following subsections describe the three classes in detail. Fig. 1 illustrates the behaviour of each class in relation to the power supply voltage. Static IC restores (R) as soon as the supply voltage v_{cc} exceeds the on-threshold of the processor, V_{ON}, and starts useful computation (C) while periodically saving snapshots at checkpoints (CP). Task-based IC also restores as soon as $v_{cc} > V_{ON}$ and features faster restore because it only needs to boot the runtime and a single task. Periodically, task-based IC saves the result of a finished task and transitions (TR) to the next task. Reactive IC sleeps until an interrupt triggers restore when v_{cc} exceeds the restore threshold V_R. Reactive IC does not take checkpoints along the way, but rather continues useful computation until v_{cc} drops below V_H, the hibernation threshold, which triggers hibernation (H), saving a snapshot just before power is lost. The circles show the latest snapshot before power is lost; any computation after the latest snapshot is a waste of resources. Note that some task-based or static approaches may avoid wasted computation after a checkpoint by measuring stored energy and comparing it to predictions [7] or observations [12] of the energy required to reach the next task-boundary or checkpoint.

2.1 Static IC

Static IC approaches are based on instrumenting application code with checkpoints by the user during application development, or automatically at compile time [7, 20]. Because the checkpoints are inserted a priori, offline analysis of program execution and control flow graphs can be applied. The main two advantages of static IC are: 1) information about program execution flow can be exploited to yield smaller snapshots [7, 14, 18, 22], and 2) snapshots are saved regardless of the environment. The former also leads to eliminating the need for energy buffering, as explained in subsection 2.3 (although static IC methods generally require some energy

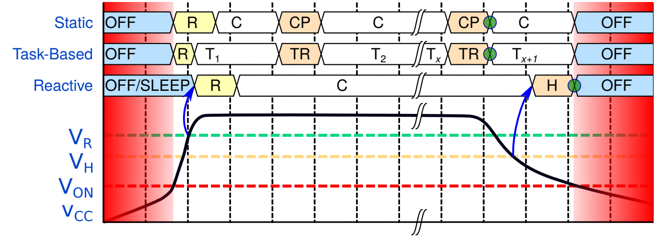

Figure 1: Behaviour of the three classes of existing IC strategies with regards to v_{cc}.

buffering for performance reasons). The latter is an advantage because no hardware to monitor the environment (e.g. supply voltage monitor) is needed. The main challenges to static IC relate to the placement of checkpoints: placing them too far apart minimises or even eliminates superfluous checkpoints, but makes it unlikely or even impossible to reach the next checkpoint; placing them too close together imposes runtime overhead because of superfluous checkpoints.

Static IC also suffers from frequent code re-execution, because the state rolls back to the latest snapshot when power returns after a power failure (the computation after the circle in Fig. 1 is re-executed in the next power cycle). Code re-execution is a waste of energy and, as observed in [19], it can lead to inconsistencies between volatile and nonvolatile memory, called idempotency violations.

2.2 Task-Based Static IC

Task-based IC [8, 12, 16] is a recent development of static IC where the application is divided into a set of tasks that are executed atomically by a runtime. The task division is motivated by protecting static IC against idempotency violations by carefully controlling each task's accesses to nonvolatile memory, thus making code re-execution safe [8]. However, re-execution is still a waste of resources. Task-based systems, like Alpaca [16] offer fast reboot and state-saving because only the runtime and the current task is booted, and only persistent data from the task needs to be saved.

The main drawback of task-based systems is the imposed programming model, requiring the programmer to redesign the application and all associated libraries. Furthermore, finding optimal task boundaries (analogous to checkpoint placement in static IC) is difficult and depends on both the specific underlying hardware and the harvesting conditions. Colin et al. recently proposed a method that checks for non-terminating path bugs[1] and performs automatic task decomposition [9]. The method appears to be well suited for simple embedded system workloads where energy consumption is deterministic. However, limitations of the method, such as the requirement that the programmer must specify an iteration bound for each unbounded loop in the program, restrict its utility for applications that depend on existing libraries.

[1] Execution paths between task boundaries which consume more energy than the device can muster [9].

2.3 Reactive IC

Contrary to static and task-based approaches, reactive IC is generally implemented in an application agnostic manner [3, 4, 11]. Reactive IC consists of the following two operations:

- **Hibernate:** Save a snapshot of volatile state to nonvolatile memory, and enter low power mode or shut down.
- **Restore:** Restore volatile state from a snapshot.

The volatile state, is any state that is lost/corrupted in the event of a power outage and which is needed for computation to proceed correctly when power returns. The set of volatile state is architecture and platform dependent.

Power supply monitoring is set up to generate interrupts that trigger hibernate and restore operations when the supply voltage v_{cc} crosses a threshold. When the restore threshold is exceeded, an interrupt triggers restore, which restores state from a previous snapshot. Similarly, when v_{cc} drops below the hibernation threshold, a snapshot is saved and the system enters low-power mode or shuts down. If power returns while the system is still in low-power mode, restore is unnecessary [3]. When powered by a low-current source, hibernation can often be avoided altogether by pre-emptively entering sleep mode before the supply voltage drops below the hibernation threshold [15]

Because reactive IC only consists of two interrupt-triggered operations, runtime overhead is minimised. The hibernation threshold ensures that snapshots are only saved when power-failure is imminent; thus nearly eliminating superfluous save operations. The restore threshold is used to guarantee the success of the next checkpoint; eliminating code re-execution, and thus also eliminating idempotency violations. Note that to guarantee the success of the next checkpoint, a minimum amount of energy buffering is required. Determining such a restore threshold is intractable for static IC because all possible execution paths must be exhaustively analysed. The main drawback of reactive IC is that saving and restoring the entire state is expensive compared to task-based systems where only one task is loaded/saved at a time. This is especially wasteful if the power cycle is short, because only a small part of the program is expected to execute.

2.4 Considerations for High-Performance IC

In high-performance systems, such as an MPSoC, application software is typically much more complex than on microcontrollers. Complications that are introduced when moving from a typical microcontroller to a high-performance MPSoC include:

(1) **Processor affinity:** Tasks/threads/processes should generally be free to migrate between (heterogeneous) cores.
(2) **Multiprocessing:** Several tasks/threads/processes may execute simultaneously.
(3) **Hierarchical memory:** Multilevel memory hierarchy (and possibly virtual memory) introduces nondeterministic delay in execution.
(4) **Dependency:** Complex software is likely to have dependencies on pre-compiled libraries.
(5) **Instruction-level parallelism:** High-performance processors typically feature superscalar pipelines that can execute more than one instruction per clock cycle. Most current implementations feature out-of-order (OoO) execution with in-order commit [17].

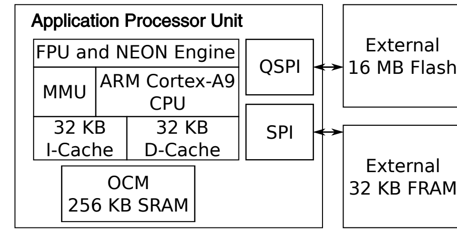

Figure 2: Processing system with external FRAM via SPI.

Processor affinity, multiprocessing and hierarchical memory all complicate path-energy estimations required for automatic insertion of task boundaries (or checkpoints); forcing suboptimal boundaries (checkpoints), assuming a set of safe boundaries (checkpoints) can be found at all. Dependency on pre-compiled libraries makes determination of the bound for the number of iterations of loops impractical, without which the automatic task decomposition such as proposed in [9] becomes inefficient (task boundaries are inserted inside every unbounded loop).

Out of order execution reduces the number of wait-cycles in the CPU when waiting for memory operations. Instructions are stalled in the pipeline until their operands are available while more instructions are fetched. Of key importance is the fact that OoO execution still commits results in program order, so from a program's perspective, the execution acts as if it was in order [17].

With reactive IC, assuming that the interrupt latency is negligible, the energy consumption of a program is irrelevant; only the energy consumption of hibernate and restore matter. The energy consumption of hibernate and restore is largely determined by the amount of state to save, and by memory performance. For the same reason, dependency on external libraries is irrelevant in reactive IC. Reactive IC can be implemented program agnostically, but at the cost of larger hibernation/restore overhead when compared to task-based IC.

3 REALISING IC ON A HIGH-PERFORMANCE PROCESSOR

To implement IC, low level access and documentation is required to safely save and restore state in an exact and verifiable manner. Therefore a Xilinx Zynq 7010 FPGA MPSoC [21] featuring a dual-core Arm A9 [1] processing system was selected as the development platform[2]. The platform was chosen because of its extensive ecosystem, allowing low-level access and debug of the A9 processing system. The FPGA fabric present in the platform was not used in this work. Fig. 2 shows a block diagram representing the system used in this work. External flash on the platform holds the application image as well as the first-stage bootloader. An external 32 kB FRAM communicating via 25MHz SPI is used for snapshot storage and the 256 kB on-chip memory (OCM) is used for the application. The FRAM is smaller than OCM, but sufficient for the application

[2]Most high-performance SoCs, like the one used in [10], restrict access to full documentation.

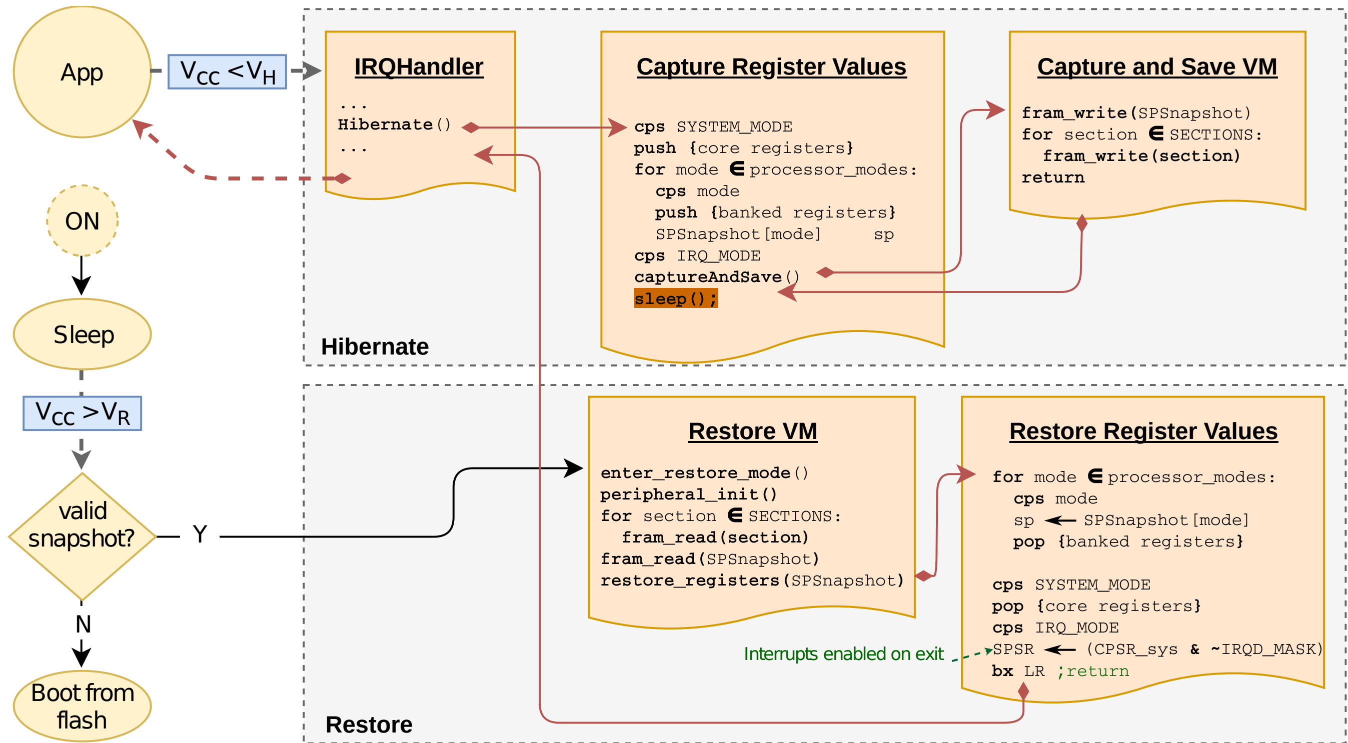

Figure 3: Boot flow and Hibernate and Restore operations.

tested in this work; larger FRAM chips are commercially available. The low-energy symmetric read/write property of FRAM makes it superior to flash for snapshot storage [4]. The development board also features DDR memory, but to use it on this specific platform requires programming of the FPGA fabric during boot and thus imposes a substantial overhead (measured at >200 ms for a minimal bitstream).

3.1 Hibernate

The hibernate operation must capture system state in a snapshot and store it in NVM. Because the system for which state is to be captured is the same system used to capture said state, it is imperative that care is taken to avoid a corrupt snapshot. We propose the following sequence of steps to ensure correctness:

(1) Capture processor state
(2) Capture co-processor state[3]
(3) Capture the state of volatile memory (VM)

Steps 1) and 2) are best handled at assembler level, while 3) is more efficiently done at a higher level of abstraction. The processor register file is captured first, so that the registers are free to be used when capturing the state of VM. Microarchitectural state such as pipeline buffers and the branch-prediction table are not saved, as they are inaccessible. Note, however, that this only affects performance, not correctness. Branch-misprediction and a flushed pipeline may cause a small latency as instructions are re-fetched immediately after continuing execution after a restore, but will not result in incorrect computation.

Depending on application complexity and resource usage, the processor and co-processor state may also include memory management unit (MMU) state, cache-controller state, etc. In this work, the core register file and the NEON (vector and floating point co-processor) register file are saved. The remaining volatile processor and co-processor state is not changed during execution, so it is more efficient to recompute them during boot than it is to save and restore them.

3.2 Restore

Restoring state is similar but opposite to hibernate. The first step is to restore the state of VM; however, before that, a context switch must be performed to avoid corrupting the restore routine. The context switch entails moving to a separate, temporary stack and data space while reading the snapshot; thus avoiding overwriting the current stack while restoring the saved one. Since an external SPI-based FRAM memory is used in this work, the SPI and associated peripherals must also be initialised before restoring from the snapshot. Data needed for the initialisation is kept in flash memory. Before returning from restore, the system must be in the exact same state as it was at the end of the hibernate routine, because restore returns to the return address of hibernate. This requires exact control of register values, and so restoring registers and returning from the restore routine must be done at the assembler level.

[3] The Arm A9 processing system includes several co-processors that control subsystems (Cache, memory management unit, preload engine etc.) or accelerate computation (data-level parallelism and floating point operations) [1].

3.3 Implementation

Fig. 3 illustrates the procedure proposed in this paper to safely hibernate and restore processor state and VM. An interrupt is triggered when v_{cc} drops below the hibernate threshold V_H. First, all general purpose registers are pushed to the stacks (there are six stacks, each belonging to a processor mode) and the stack pointers are saved in an array `SPSnapshot`. All processor modes are looped through in order to capture the banked registers belonging to each mode. `SPSnapshot` is then written to NVM before proceeding to copy allocated memory to NVM. The processor then enters a low-power mode.

When power returns, the processor immediately sleeps until v_{cc} exceeds the restore threshold V_R. Determining the thresholds V_R and V_H is not discussed in this paper; an efficient approach that is applicable to the system presented in this work is found in [3].

If no valid snapshot is present in FRAM (such as when booting up for the first time), the system boots from flash. If a valid snapshot is present, a context switch is performed (`enter_restore_mode()`) to avoid corruption of the stack. Then peripherals, including the SPI peripheral used to read the snapshot, are initialised using default configuration held in flash. All allocated memory is then loaded from the snapshot, followed by reading the saved stack pointers into `SPSnapshot`. Each processor mode is looped through to pop the core registers off each stack after restoring the stack pointer from `SPSnapshot`. The processor is switched into IRQ mode (`cps IRQ_MODE`) and the SPSR register is set up such that interrupts are enabled upon returning through the interrupt routine. Finally, a branch to the restored link register returns through the interrupt routine as if execution continued after hibernate was called in the previous power cycle.

The method presented in this paper can be adapted to any processing system *without regard for the application*. Once the method is implemented on a specific platform, that platform can run any application, based on any code base, on an intermittent supply.

3.4 Multiprocessing

The method presented in this section can be extended for multiprocessor systems as shown in Fig. 4. When a hibernation/restore interrupt occurs, each processor begins executing its hibernation/restore routine. One core is selected (arbitrarily) as the primary core. When hibernating, all cores save their own processor state. All but the primary core then sleep; the primary core saves the state of VM and any common processor state (such as co-processor state) before entering sleep mode. When restoring from a snapshot, all but the primary core stall. The primary core restores VM and any common processor state before notifying the other cores. Then all cores restore their processor state and resume execution from the snapshot.

4 TAILORING HIGH-PERFORMANCE SYSTEMS TOWARDS EFFICIENT IC

If IC is to be viable on a high-performance system, it is crucial to optimise for fast reboots. Typically memory allocation and the boot process is geared towards optimising runtime performance because traditional computers rarely reboot. This section presents general concepts and optimisations to tailor high-performance system towards more efficient IC.

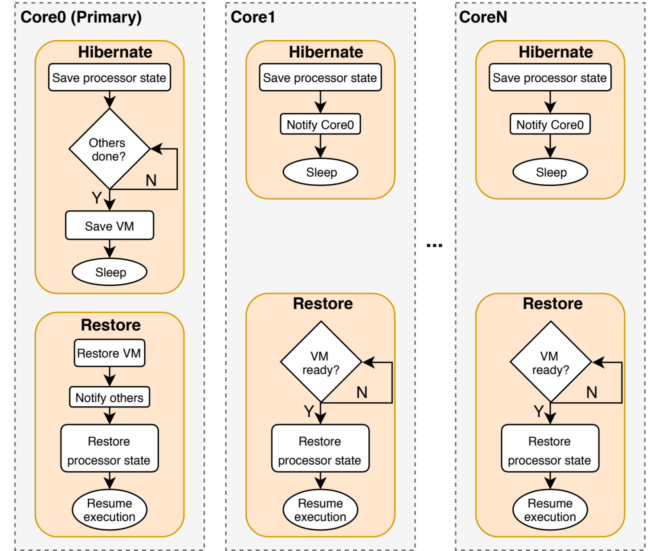

Figure 4: Hibernation and restore procedure for multiprocessor systems.

Boot process. Traditional computers load both modifiable and constant sections to main memory (typically DDR) before execution. This results in better runtime performance because main memory offers orders of magnitude faster access time than non-volatile storage (flash/disk). However, in IC only a small portion of an application is expected to execute in a single power cycle; it is therefore a waste of resources to load the entire program. Execute-in-place (XIP), where instructions are executed directly from flash, is a better solution for IC, because only modifiable sections (`.data`, `.bss`) are loaded during boot. The instruction cache aleviates excessively repeated reads of the same instruction from flash for most applications.

Memory Allocation. In traditional computing, allocating uninitialized data to the `.bss` section reduces the size of the application's binary image and reduces the amount of data loaded during boot. For IC, the `.bss` section has little utility beyond legacy; `.bss` must persist through power cycles and so it must still be loaded during boot and it still requires space in the snapshot. Some data is more efficient to re-initialize/re-compute than to save/restore to/from NVM; especially when the NVM is severely slower than main memory. We propose a new, IC specific, section `.npbss` (non-persistent bss) that is not included in the snapshot; and hence never saved/restored. This section is useful for variables needed in the booting process or to initialise peripherals. One particular example in this work was the configuration structure for the Generic Interrupt Controller (GIC) which requires a large table of pointers to, and arguments for, interrupt handlers. Allocating the GIC structure to `.npbss` in place of `.bss` reduced the snapshot size by 1572 bytes (10% of total snapshot size), thus significantly reducing the data written/read during hibernate/restore.

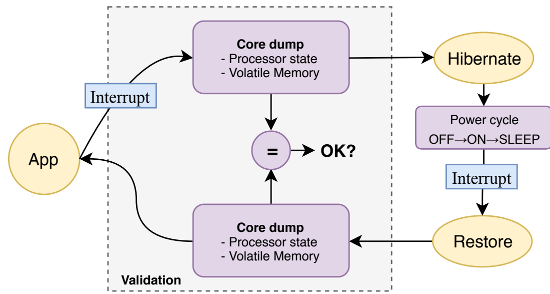

Figure 5: Validation of correctly saved and restored state.

5 EXPERIMENTAL VALIDATION AND ANALYSIS

This work has found and extended the most suitable strategy for IC on high-performance MPSoC systems. In this section, we validate the implementation on an A9 processing system residing on a Xilinx Zynq 7010 FPGA MPSoC [21]. Practical multiprocessing requires an operating system with multiprocessing support, such as Linux. However, these operating systems are unsuitable for IC due to their resource usage, and must be adapted for efficient IC. The chosen IC strategy was therefore implemented and validated on a single processing core.

5.1 Boot Process

Boot time is critical for the viability of IC. For the purpose of this work the boot time can be divided in three parts:

- $t_{HWSETUP}$ – hardware setup time, i.e. the time taken from $v_{cc} > V_{ON}$ until the first instruction of the bootloader is executed
- t_{BOOT0} – execution time of the 0th stage bootloader
- t_{FSBL} – execution time of the first-stage bootloader (FSBL)

BOOT0 is a ROM bootloader programmed during fabrication that loads FSBL from the external flash (Fig. 2) to on-chip memory. FSBL loads the application from external flash to main memory and also programs the FPGA fabric if a bitstream is present in the flash image. In this work, OCM was used as main memory to avoid programming of the FPGA fabric, making the boot process representative of a conventional application processor.

To minimise t_{BOOT0}, FSBL was modified and compiled to be executed directly from flash rather than loaded into OCM, saving 9*ms* of boot time. Because BOOT0 cannot be instrumented for timing measurements, $t_{HWSETUP}$ and t_{BOOT0} were combined into $t_{RST\rightarrow FSBL} = t_{HWSETUP} + t_{BOOT0}$ which is readily measured by instrumentation of FSBL. An output pin at was set the start of FSBL and the time between $v_{cc} > V_{ON}$ and the pin transitioning to logic "1" was measured with an oscilloscope to be 260 *ms*. In a similar manner, t_{FSBL} was measured as 19.5 *ms*.

While there is room for improvement in t_{FSBL}, $t_{RST\rightarrow FSBL}$ cannot be improved further beyond using XIP and minimising the amount of modifiable data loaded for FSBL. This presents a strong limitation in the viability of this platform being used for IC.

5.2 Validation of Hibernate and Restore

The system shown in Fig. 3 was implemented on the Zynq SoC. Validation was performed using an FFT of 300 complex floating point numbers as a sample application. For simplicity of validation, hibernate and restore was triggered at random points in the program by use of IO interrupts rather than voltage monitoring.

To verify correctness of the snapshot, core dumps were used to capture volatile state immediately before hibernate and after restore, as shown in Fig. 5. The power supply was cycled between hibernate and restore to ensure that all volatile state was lost before restoring. The core dumps were taken by use of breakpoints and the memory dump function of the debugger. The allocated portion of the core dumps were then verified as being equal, showing that state is consistent through power cycles.

Finally, the performance of hibernate and restore was measured. The time consumed was measured by an on-chip timer and was measured from the start of the interrupt routine until a snapshot had been successfully saved/restored. The restore and hibernate times were both measured to be 8.6 *ms* with a snapshot size of 15 *kB*.

6 CONCLUSIONS AND FUTURE WORK

Motivated by recent developments in power-neutral computing, this paper surveyed published research in intermittent computing to determine the most suitable approach for implementing a zero-power state for power-neutral high-performance MPSoCs. Reactive IC was found to be the most viable option based on general properties of MPSoCs and their implications to existing work. Techniques to tailor memory allocation and the boot process towards more efficient IC were then presented. Reactive IC was then implemented on an Arm A9 processor and experimentally validated. A method to validate consistency of state through power cycles was presented. State was correctly saved and restored by implementing two program-agnostic functions, hibernate and restore.

The time taken to hibernate and the time taken to restore were both measured to be 8.6 *ms* for the tested application, which had 15 *kB* of volatile state (snapshot size). Experiments measuring boot time showed that the time from power reset to the start of execution of the first-stage bootloader was 260 *ms*; this presents the minimum bound on start-up time for the specific hardware platform used in this work. Future research on this topic should use an SoC with faster start-up. The current method for power-neutral performance scaling of MPSoCs is based on the Linux operating system. Future work may explore whether Linux can be tailored towards IC or whether other operating systems can fill the gap. When a suitable operating system has been found, the multiprocessing extensions proposed in this work should be validated on a multicore system.

ACKNOWLEDGMENTS

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) under an iCASE award and Grant EP/P010164/1, and Grant EP/K034448/1. No data except those explicitly stated in the paper were generated during the study.

REFERENCES

[1] Arm. 2011. Cortex-A9 Technical Reference Manual. (2011).

[2] Domenico Balsamo, Anup Das, Alex S. Weddell, Davide Brunelli, Bashir M. Al-Hashimi, Geoff V. Merrett, and Luca Benini. 2016. Graceful Performance Modulation for Power-Neutral Transient Computing Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 5 (2016), 738–749. https://doi.org/10.1109/TCAD.2016.2527713

[3] Domenico Balsamo, Alex S Weddell, Anup Das, Geoff V Merrett, Bashir M Al-hashimi, Davide Brunelli, and Luca Benini. 2016. Hibernus++ : A Self-Calibrating and Adaptive System for Intermittently-Powered Embedded Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 12 (2016), 1968–1980. https://doi.org/10.5258/SOTON/389749

[4] Domenico Balsamo, Alex S. Weddell, Geoff V. Merrett, Bashir M. Al-Hashimi, Davide Brunelli, and Luca Benini. 2015. Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems. *IEEE Embedded Systems Letters* 7, 1 (2015), 15–18. https://doi.org/10.1109/LES.2014.2371494

[5] Stephen Beeby and Neil White. 2010. *Energy Harvesting for Autonomous Systems*. Artech House, Norwood, MA.

[6] Naveed Anwar Bhatti and Luca Mottola. 2016. Efficient State Retention for Transiently-powered Embedded Sensing. In *EWSN '16 Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*. 137–148.

[7] Naveed Anwar Bhatti and Luca Mottola. 2017. HarvOS: Efficient Code Instrumentation for Transiently-powered Embedded Sensing. In *ACM/IEEE International Conference on Information Processing in Sensor Networks*. Pittsburgh, PA, USA, 209–219. https://doi.org/10.1145/3055031.3055082

[8] Alexei Colin and Brandon Lucia. 2016. Chain: tasks and channels for reliable intermittent programs. In *ACM Special Interest Group on Programming Languages Notices*, Vol. 51. Amsterdam, Netherlands, 514–530. https://doi.org/10.1145/3022671.2983995

[9] Alexei Colin and Brandon Lucia. 2018. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction - CC 2018*. 116–127. https://doi.org/10.1145/3178372.3179525

[10] B. J. Fletcher, D. Balsamo, and G. V. Merrett. 2017. Power neutral performance scaling for energy harvesting MP-SoCs. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 1516–1521. https://doi.org/10.23919/DATE.2017.7927231

[11] Hrishikesh Jayakumar, Arnab Raha, Woo Suk Lee, and Vijay Raghunathan. 2015. QuickRecall: A HW/SW Approach for Computing Across Power Cycles in Transiently Powered Computers. *J. Emerg. Technol. Comput. Syst.* 12, 1, Article 8 (Aug. 2015), 19 pages. https://doi.org/10.1145/2700249

[12] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. 2016. Energy-Aware Memory Mapping for Hybrid FRAM-SRAM MCUs in IoT Edge Devices. In *Proceedings of the IEEE International Conference on VLSI Design*, Vol. 2016-March. 264–269. https://doi.org/10.1109/VLSID.2016.52

[13] Aman Kansal, Jason Hsu, Sadaf Zahedi, and Mani B. Srivastava. 2007. Power management in energy harvesting sensor networks. *ACM Transactions on Embedded Computing Systems* 6, 4 (2007). https://doi.org/10.1145/1274858.1274870

[14] Qingan Li, Mengying Zhao, Jingtong Hu, Yongpan Liu, Yanxiang He, and Chun Jason Xue. 2015. Compiler directed automatic stack trimming for efficient non-volatile processors. *Proceedings of the 52nd Annual Design Automation Conference on - DAC '15* 299 (2015), 1–6. https://doi.org/10.1145/2744769.2744809

[15] Giedrius Lukosevicius, Alberto Rodriguez Arreola, and Alex S Weddell. 2017. Using Sleep States to Maximize the Active Time of Transient Computing Systems. In *Proceedings of the Fifth ACM International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems - ENSsys'17*. 31–36. https://doi.org/10.1145/3142992.3142998

[16] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: intermittent execution without checkpoints. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30. https://doi.org/10.1145/3133920

[17] David A. Patterson and John L. Hennesey. 2016. *Computer Organization And Design* (1 ed.). Morgan Kaufmann.

[18] J S Plank, M Beck, and G Kingsley. 1995. Compiler-Assisted Memory Exclusion for Fast Checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments* 7, 4 (1995), 10–14.

[19] Benjamin Ransford and Brandon Lucia. 2014. Nonvolatile memory is a broken time machine. In *Proceedings of the workshop on Memory Systems Performance and Correctness, MSPC'14*. https://doi.org/10.1145/2618128.2618136

[20] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System support for long-running computation on RFID-scale devices. In *ACM ASPLOS*. Newport Beach, CA, USA, 159–170. https://doi.org/10.1145/1950365.1950386

[21] Xilinx. 2013. UG585 Zynq-7000 All Programmable SoC TRM. (2013).

[22] Mengying Zhao, Qingan Li, Mimi Xie, Yongpan Liu, Jingtong Hu, and Chun Jason Xue. 2015. Software Assisted Non-volatile Register Reduction for Energy Harvesting Based Cyber-physical System. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE '15)*. EDA Consortium, San Jose, CA, USA, 567–572. http://dl.acm.org/citation.cfm?id=2755753.2755881